下载APP

35｜即学即练：如何实现一个轻量级线程池？

2022-01-17 Tony Bai

《Tony Bai · Go语言第一课》 课程介绍 >

讲述：Tony Bai
时长 16:22 大小 15.00M

你好，我是 Tony Bai。

在这一讲的开始，首先恭喜你完成了这门课核心篇语法部分的学习。这一部分的篇幅不多，主要讲解了 Go 的两个核心语法知识点：接口与并发原语。它们分别是耦合设计与并发设计的主要参与者，Go 应用的骨架设计离不开它们。

领资料

但理论和实践毕竟是两回事，学完了基本语法，也需要实操来帮助我们落地。所以，在这核心篇的最后一讲，我依然会用一个小实战项目，帮助你学会灵活运用这部分的语法点

不过，关于接口类型做为“关节”作用的演示，我们前面的两个小实战项目中都有一定的体现了，只是那时还没有讲到接口类型，你现在可以停下来，回顾一下09 讲和27 讲的代码，看看是否有更深刻的体会。

而且，接口类型对 Go 应用静态骨架的编织作用，在接口类型数量较多的项目中体现得更明显，由于篇幅有限，我很难找到一个合适的演示项目。

因此，这一讲的实战项目，我们主要围绕 Go 并发来做，实现一个轻量级线程池，也就是 Goroutine 池。

为什么要用到 Goroutine 池？

在第 31 讲学习 Goroutine 的时候，我们就说过：相对于操作系统线程，Goroutine 的开销十分小，一个 Goroutine 的起始栈大小为 2KB，而且创建、切换与销毁的代价很低，我们可以创建成千上万甚至更多 Goroutine。

所以和其他语言不同的是，Go 应用通常可以为每个新建立的连接创建一个对应的新 Goroutine，甚至是为每个传入的请求生成一个 Goroutine 去处理。这种设计还有一个好处，实现起来十分简单，Gopher 们在编写代码时也没有很高的心智负担。

不过，Goroutine 的开销虽然“廉价”，但也不是免费的。

最明显的，一旦规模化后，这种非零成本也会成为瓶颈。我们以一个 Goroutine 分配 2KB 执行栈为例，100w Goroutine 就是 2GB 的内存消耗。

其次，Goroutine 从 Go 1.4 版本开始采用了连续栈的方案，也就是每个 Goroutine 的执行栈都是一块连续内存，如果空间不足，运行时会分配一个更大的连续内存空间作为这个 Goroutine 的执行栈，将原栈内容拷贝到新分配的空间中来。

连续栈的方案，虽然能避免 Go 1.3 采用的分段栈会导致的“hot split”问题，但连续栈的原理也决定了，一旦 Goroutine 的执行栈发生了 grow，那么即便这个 Goroutine 不再需要那么大的栈空间，这个 Goroutine 的栈空间也不会被 Shrink（收缩）了，这些空间可能会处于长时间闲置的状态，直到 Goroutine 退出。

另外，随着 Goroutine 数量的增加，Go 运行时进行 Goroutine 调度的处理器消耗，也会随之增加，成为阻碍 Go 应用性能提升的重要因素。

那么面对这样的问题，常见的应对方式是什么呢？

Goroutine 池就是一种常见的解决方案。这个方案的核心思想是对 Goroutine 的重用，也就是把 M 个计算任务调度到 N 个 Goroutine 上，而不是为每个计算任务分配一个独享的 Goroutine，从而提高计算资源的利用率。

接下来，我们就来真正实现一个简单的 Goroutine 池，我们叫它 **workerpool**。

workerpool 的实现原理

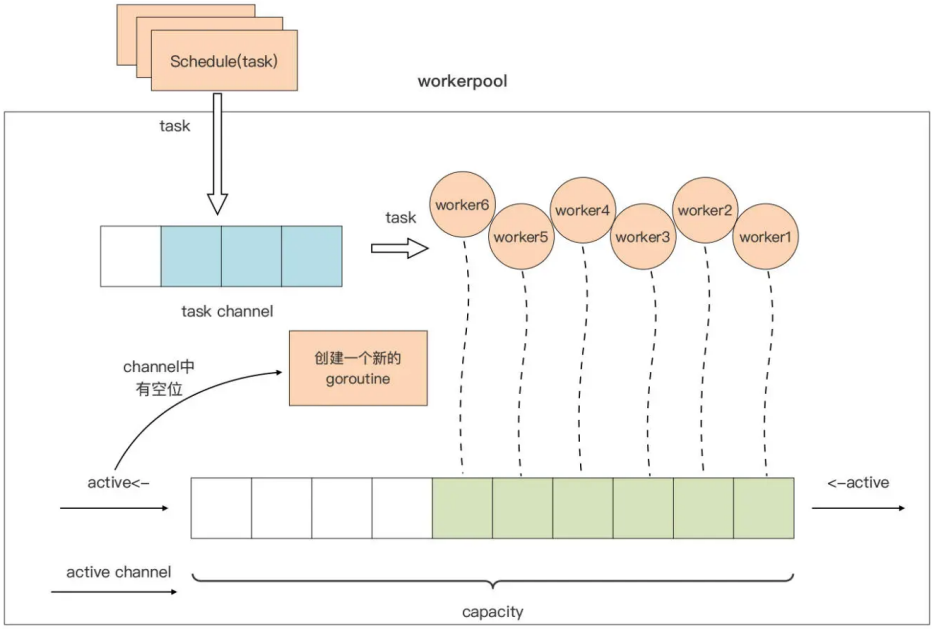
workerpool 的工作逻辑通常都很简单，所以即便是用于生产环境的 workerpool 实现，代码规模也都在千行左右。

当然，workerpool 有很多种实现方式，这里为了更好地演示 Go 并发模型的应用模式，以及并发原语间的协作，我们采用完全基于 channel+select 的实现方案，不使用其他数据结构，也不使用 sync 包提供的各种同步结构，比如 Mutex、RWMutex，以及 Cond 等。

workerpool 的实现主要分为三个部分：

- pool 的创建与销毁；
- pool 中 worker（Goroutine）的管理；
- task 的提交与调度。

其中，后两部分是 pool 的“精髓”所在，这两部分的原理我也用一张图表示了出来：



极客时间

我们先看一下图中 pool 对 worker 的管理。

capacity 是 pool 的一个属性，代表整个 pool 中 worker 的最大容量。我们使用一个带缓冲的 channel：active，作为 worker 的“计数器”，这种 channel 使用模式就是我们在第 33 讲中讲过的**计数信号量**，如果记不太清了可以复习一下第 33 讲中的相关内容。

当 active channel 可写时，我们就创建一个 worker，用于处理用户通过 Schedule 函数提交的待处理的请求。当 active channel 满了的时候，pool 就会停止 worker 的创建，直到某个 worker 因故退出，active channel 又空出一个位置时，pool 才会创建新的 worker 填补那个空位。

这张图里，我们把用户要提交给 workerpool 执行的请求抽象为一个 Task。Task 的提交与调度也很简单：Task 通过 Schedule 函数提交到一个 task channel 中，已经创建的 worker 将从这个 task channel 中读取 task 并执行。

好了！“Talk is cheap，show me the code”！接下来，我们就来写一版 workerpool 的代码，来验证一下这里分析的原理是否可行。

workerpool 的一个最小可行实现

我们先建立 workerpool 目录作为实战项目的源码根目录，然后为这个项目创建 go module：

复制代码

```
$mkdir workerpool1
$cd workerpool1
$go mod init github.com/bigwhite/workerpool
```

接下来，我们创建 pool.go 作为 workpool 包的主要源码文件。在这个源码文件中，我们定义了 Pool 结构体类型，这个类型的实例代表一个 workerpool：

复制代码

```
type Pool struct {
    capacity int          // workerpool大小

    active chan struct{} // 对应上图中的active channel
    tasks  chan Task     // 对应上图中的task channel

    wg   sync.WaitGroup  // 用于在pool销毁时等待所有worker退出
    quit chan struct{}   // 用于通知各个worker退出的信号channel
}
```

workerpool 包对外主要提供三个 API，它们分别是：

workerpool.New：用于创建一个 pool 类型实例，并将 pool 池的 worker 管理机制运行起来；

workerpool.Free：用于销毁一个 pool 池，停掉所有 pool 池中的 worker；

Pool.Schedule：这是 Pool 类型的一个导出方法，workerpool 包的用户通过该方法向 pool 池提交待执行的任务（Task）。

接下来我们就重点看看这三个 API 的实现。

我们先来看看 workerpool.New 是如何创建一个 pool 实例的：

复制代码

```
func New(capacity int) *Pool {
    if capacity <= 0 {
```

```
        capacity = defaultCapacity
    }
    if capacity > maxCapacity {
        capacity = maxCapacity
    }

    p := &Pool{
        capacity: capacity,
        tasks:    make(chan Task),
        quit:     make(chan struct{}),
        active:   make(chan struct{}, capacity),
    }

    fmt.Printf("workerpool start\n")

    go p.run()

    return p
}
```

我们看到，New 函数接受一个参数 capacity 用于指定 workerpool 池的容量，这个参数用于控制 workerpool 最多只能有 capacity 个 worker，共同处理用户提交的任务请求。函数开始处有一个对 capacity 参数的“防御性”校验，当用户传入不合理的值时，函数 New 会将它纠正为合理的值。

Pool 类型实例变量 p 完成初始化后，我们创建了一个新的 Goroutine，用于对 workerpool 进行管理，这个 Goroutine 执行的是 Pool 类型的 run 方法：

复制代码

```
func (p *Pool) run() {
    idx := 0

    for {
        select {
        case <-p.quit:
            return
        case p.active <- struct{}{}:
            // create a new worker
            idx++
            p.newWorker(idx)
        }
    }
}
```

run 方法内是一个无限循环，循环体中使用 select 监视 Pool 类型实例的两个 channel：quit 和 active。这种在 for 中使用 select 监视多个 channel 的实现，在 Go 代码中十分常见，是一种惯用法。

当接收到来自 quit channel 的退出“信号”时，这个 Goroutine 就会结束运行。而当 active channel 可写时，run 方法就会创建一个新的 worker Goroutine。 此外，为了方便在程序中区分各个 worker 输出的日志，我这里将一个从 1 开始的变量 idx 作为 worker 的编号，并把它以参数的形式传给创建 worker 的方法。

我们再将创建新的 worker goroutine 的职责，封装到一个名为 newWorker 的方法中：

复制代码

```
func (p *Pool) newWorker(i int) {
    p.wg.Add(1)
    go func() {
        defer func() {
            if err := recover(); err != nil {
                fmt.Printf("worker[%03d]: recover panic[%s] and exit\n", i, er
                <-p.active
            }
            p.wg.Done()
        }()

        fmt.Printf("worker[%03d]: start\n", i)

        for {
            select {
            case <-p.quit:
                fmt.Printf("worker[%03d]: exit\n", i)
                <-p.active
                return
            case t := <-p.tasks:
                fmt.Printf("worker[%03d]: receive a task\n", i)
                t()
            }
        }
    }()
}
```

我们看到，在创建一个新的 worker goroutine 之前，newWorker 方法会先调用 p.wg.Add 方法将 WaitGroup 的等待计数加一。由于每个 worker 运行于一个独立的

Goroutine 中，newWorker 方法通过 go 关键字创建了一个新的 Goroutine 作为 worker。

新 worker 的核心，依然是一个基于 for-select 模式的循环语句，在循环体中，新 worker 通过 select 监视 quit 和 tasks 两个 channel。和前面的 run 方法一样，当接收到来自 quit channel 的退出“信号”时，这个 worker 就会结束运行。tasks channel 中放置的是用户通过 Schedule 方法提交的请求，新 worker 会从这个 channel 中获取最新的 Task 并运行这个 Task。

Task 是一个对用户提交的请求的抽象，它的本质就是一个函数类型：

复制代码

```
type Task func()
```

这样，用户通过 Schedule 方法实际上提交的是一个函数类型的实例。

在新 worker 中，为了防止用户提交的 task 抛出 panic，进而导致整个 workerpool 受到影响，我们在 worker 代码的开始处，使用了 defer+recover 对 panic 进行捕捉，捕捉后 worker 也是要退出的，于是我们还通过<-p.active更新了 worker 计数器。并且一旦 worker goroutine 退出，p.wg.Done 也需要被调用，这样可以减少 WaitGroup 的 Goroutine 等待数量。

我们再来看 workerpool 提供给用户提交请求的导出方法 Schedule：

复制代码

```
var ErrWorkerPoolFreed    = errors.New("workerpool freed")        // workerpool

func (p *Pool) Schedule(t Task) error {
    select {
    case <-p.quit:
        return ErrWorkerPoolFreed
    case p.tasks <- t:
        return nil
    }
}
```

Schedule 方法的核心逻辑，是将传入的 Task 实例发送到 workerpool 的 tasks channel 中。但考虑到现在 workerpool 已经被销毁的状态，我们这里通过一个 select，检视 quit channel 是否有“信号”可读，如果有，就返回一个哨兵错误 ErrWorkerPoolFreed。如果没有，一旦 p.tasks 可写，提交的 Task 就会被写入 tasks channel，以供 pool 中的 worker 处理。

这里要注意的是，这里的 Pool 结构体中的 tasks 是一个无缓冲的 channel，如果 pool 中 worker 数量已达上限，而且 worker 都在处理 task 的状态，那么 Schedule 方法就会阻塞，直到有 worker 变为 idle 状态来读取 tasks channel，schedule 的调用阻塞才会解除。

至此，workerpool 的最小可行实现的主要逻辑都实现完了。我们来验证一下它是否能按照我们的预期逻辑运行。

现在我们建立一个使用 workerpool 的项目 demo1：

复制代码

```
$mkdir demo1
$cd demo1
$go mod init demo1
```

由于我们要引用本地的 module，所以我们需要手工修改一下 demo1 的 go.mod 文件，并利用 replace 指示符将 demo1 对 workerpool 的引用指向本地 workerpool1 路径：

复制代码

```
module demo1

go 1.17

require github.com/bigwhite/workerpool v1.0.0

replace github.com/bigwhite/workerpool v1.0.0 => ../workerpool1
```

然后创建 demo1 的 main.go 文件，源码如下：

复制代码

```go
package main

import (
    "time"
    "github.com/bigwhite/workerpool"
)

func main() {
    p := workerpool.New(5)

    for i := 0; i < 10; i++ {
        err := p.Schedule(func() {
            time.Sleep(time.Second * 3)
        })
        if err != nil {
            println("task: ", i, "err:", err)
        }
    }

    p.Free()
}
```

这个示例程序创建了一个 capacity 为 5 的 workerpool 实例，并连续向这个 workerpool 提交了 10 个 task，每个 task 的逻辑很简单，只是 Sleep 3 秒后就退出。main 函数在提交完任务后，调用 workerpool 的 Free 方法销毁 pool，pool 会等待所有 worker 执行完 task 后再退出。

demo1 示例的运行结果如下：

复制代码

```
workerpool start
worker[005]: start
worker[005]: receive a task
worker[003]: start
worker[003]: receive a task
worker[004]: start
worker[004]: receive a task
worker[001]: start
worker[002]: start
worker[001]: receive a task
worker[002]: receive a task
worker[004]: receive a task
worker[005]: receive a task
worker[003]: receive a task
worker[002]: receive a task
worker[001]: receive a task

```

```
worker[001]: exit
worker[005]: exit
worker[002]: exit
worker[003]: exit
worker[004]: exit
workerpool freed
```

从运行的输出结果来看，workerpool 的最小可行实现的运行逻辑与我们的原理图是一致的。

不过，目前的 workerpool 实现好比“铁板一块”，虽然我们可以通过 capacity 参数可以指定 workerpool 容量，但我们无法对 workerpool 的行为进行定制。

比如当 workerpool 中的 worker 数量已达上限，而且 worker 都在处理 task 时，用户调用 Schedule 方法将阻塞，如果用户不想阻塞在这里，以我们目前的实现是做不到的。

那我们可以怎么改进呢？我们可以尝试在上面实现的基础上，为 workerpool 添加功能选项（functional option）机制。

添加功能选项机制

功能选项机制，可以让某个包的用户可以根据自己的需求，通过设置不同功能选项来定制包的行为。Go 语言中实现功能选项机制有多种方法，但 Go 社区目前使用最为广泛的一个方案，是 Go 语言之父 Rob Pike 在 2014 年在博文《自引用函数与选项设计》中论述的一种，这种方案也被后人称为“功能选项（functional option）”方案。

接下来，我们就来看看如何使用 Rob Pike 的这种“功能选项”方案，让 workerpool 支持行为定制机制。

首先，我们将 workerpool1 目录拷贝一份形成 workerpool2 目录，我们将在这个目录下为 workerpool 包添加功能选项机制。

然后，我们在 workerpool2 目录下创建 option.go 文件，在这个文件中，我们定义用于代表功能选项的类型 Option：

复制代码

```
type Option func(*Pool)
```

我们看到，这个 Option 实质是一个接受 *Pool 类型参数的函数类型。那么如何运用这个 Option 类型呢？别急，马上你就会知道。现在我们先要做的是，明确给 workerpool 添加什么功能选项。这里我们为 workerpool 添加两个功能选项：Schedule 调用是否阻塞，以及是否预创建所有的 worker。

为了支持这两个功能选项，我们需要在 Pool 类型中增加两个 bool 类型的字段，字段的具体含义，我也在代码中注释了：

复制代码

```
type Pool struct {
    ... ...
    preAlloc bool // 是否在创建pool的时候就预创建workers，默认值为：false

    // 当pool满的情况下，新的Schedule调用是否阻塞当前goroutine。默认值：true
    // 如果block = false，则Schedule返回ErrNoWorkerAvailInPool
    block  bool
    ... ...
}
```

针对这两个字段，我们在 option.go 中添加两个功能选项，WithBlock 与 WithPreAllocWorkers：

复制代码

```
func WithBlock(block bool) Option {
    return func(p *Pool) {
        p.block = block
    }
}

func WithPreAllocWorkers(preAlloc bool) Option {
    return func(p *Pool) {
        p.preAlloc = preAlloc
    }
}
```

我们看到，这两个功能选项实质上是两个返回闭包函数的函数。

为了支持将这两个 Option 传给 workerpool，我们还需要改造一下 workerpool 包的 New 函数，改造后的 New 函数代码如下：

复制代码

```
func New(capacity int, opts ...Option) *Pool {
    ... ...
    for _, opt := range opts {
        opt(p)
    }

    fmt.Printf("workerpool start(preAlloc=%t)\n", p.preAlloc)

    if p.preAlloc {
        // create all goroutines and send into works channel
        for i := 0; i < p.capacity; i++ {
            p.newWorker(i + 1)
            p.active <- struct{}{}
        }
    }

    go p.run()

    return p
}
```

新版 New 函数除了接受 capacity 参数之外，还在它的参数列表中增加了一个类型为 Option 的可变长参数 opts。在 New 函数体中，我们通过一个 for 循环，将传入的 Option 运用到 Pool 类型的实例上。

新版 New 函数还会根据 preAlloc 的值来判断是否预创建所有的 worker，如果需要，就调用 newWorker 方法把所有 worker 都创建出来。newWorker 的实现与上一版代码并没有什么差异，这里就不再详说了。

但由于 preAlloc 选项的加入，Pool 的 run 方法的实现有了变化，我们来看一下：

复制代码

```
 func (p *Pool) run() {
     idx := len(p.active)

     if !p.preAlloc {
     loop:
         for t := range p.tasks {
```

```
            p.returnTask(t)
            select {
            case <-p.quit:
                return
            case p.active <- struct{}{}:
                idx++
                p.newWorker(idx)
            default:
                break loop
            }
        }
    }

    for {
        select {
        case <-p.quit:
            return
        case p.active <- struct{}{}:
            // create a new worker
            idx++
            p.newWorker(idx)
        }
    }
}
```

新版 run 方法在 preAlloc=false 时，会根据 tasks channel 的情况在适合的时候创建 worker（第 4 行~ 第 18 行），直到 active channel 写满，才会进入到和第一版代码一样的调度逻辑中（第 20 行~ 第 29 行）。

而且，提供给用户的 Schedule 函数也因 WithBlock 选项，有了一些变化：

复制代码

```
func (p *Pool) Schedule(t Task) error {
    select {
    case <-p.quit:
        return ErrWorkerPoolFreed
    case p.tasks <- t:
        return nil
    default:
        if p.block {
            p.tasks <- t
            return nil
        }
        return ErrNoIdleWorkerInPool
    }
}
```

Schedule 在 tasks chanel 无法写入的情况下，进入 default 分支。在 default 分支中，Schedule 根据 block 字段的值，决定究竟是继续阻塞在 tasks channel 上，还是返回 ErrNoIdleWorkerInPool 错误。

和第一版 worker 代码一样，我们也来验证一下新增的功能选项是否好用。我们建立一个使用新版 workerpool 的项目 demo2，demo2 的 go.mod 与 demo1 的 go.mod 相似：

复制代码

```
module demo2

go 1.17

require github.com/bigwhite/workerpool v1.0.0

replace github.com/bigwhite/workerpool v1.0.0 => ../workerpool2
```

demo2 的 main.go 文件如下：

复制代码

```
package main

import (
    "fmt"
    "time"

    "github.com/bigwhite/workerpool"
)

func main() {
    p := workerpool.New(5, workerpool.WithPreAllocWorkers(false), workerpool.W

    time.Sleep(time.Second * 2)
    for i := 0; i < 10; i++ {
        err := p.Schedule(func() {
            time.Sleep(time.Second * 3)
        })
        if err != nil {
            fmt.Printf("task[%d]: error: %s\n", i, err.Error())
        }
    }

    p.Free()
}
```

在 demo2 中，我们使用 workerpool 包提供的功能选项，设置了我们期望的 workerpool 的运作行为，包括不要预创建 worker，以及不要阻塞 Schedule 调用。

考虑到 Goroutine 调度的次序的不确定性，这里我在创建 workerpool 与真正开始调用 Schedule 方法之间，做了一个 Sleep，尽量减少 Schedule 都返回失败的频率（但这仍然无法保证这种情况不会发生）。

运行 demo2，我们会得到这个结果：

复制代码

```
workerpool start(preAlloc=false)
task[1]: error: no idle worker in pool
worker[001]: start
task[2]: error: no idle worker in pool
task[4]: error: no idle worker in pool
task[5]: error: no idle worker in pool
task[6]: error: no idle worker in pool
task[7]: error: no idle worker in pool
task[8]: error: no idle worker in pool
task[9]: error: no idle worker in pool
worker[001]: receive a task
worker[002]: start
worker[002]: exit
worker[001]: receive a task
worker[001]: exit
workerpool freed(preAlloc=false)
```

不过，由于 Goroutine 调度的不确定性，这个结果仅仅是很多种结果的一种。我们看到，仅仅 001 这个 worker 收到了 task，其余的 worker 都因为 worker 尚未创建完毕，而返回了错误，而不是像 demo1 那样阻塞在 Schedule 调用上。

小结

好了，今天的课讲到这里就结束了，现在我们一起来回顾一下吧。

在这一讲中，我们基于我们前面所讲的 Go 并发方面的内容，设计并实现了一个 workerpool 的最小可行实现，只用了不到 200 行代码。为了帮助你理解 Go 并发原语是

如何运用的，这个 workerpool 实现完全基于 channel+select，并没有使用到 sync 包提供的各种锁。

我们还基于 workerpool 的最小可行实现，为这个 pool 增加了功能选项的支持，我们采用的功能选项方案也是 Go 社区最为流行的方案，日常编码中如果你遇到了类似的需求可以重点参考。

最后我要提醒你：上面设计与实现的 workerpool 只是一个演示项目，不能作为生产项目使用。

思考题

关于 workerpool 这样的项目，如果让你来设计，你的设计思路是什么，不妨在留言区敞开谈谈？

欢迎你把这节课分享给更多感兴趣的朋友。我是 Tony Bai，我们下节课见。

今天的项目源码在这里！

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独订阅本课程，你将得 20 元

生成海报并分享

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 34 | 并发：如何使用共享变量？

下一篇 加餐 | 如何拉取私有的Go Module？

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取

精选留言 (6)

写留言

罗杰 神

2022-01-18

实战项目棒棒哒

作者回复: 👍

1

$侯

2022-01-17

老师您好请教几个问题：
第一个问题，demo1中没有看到p.Free的代码示例，Free方法只是向p.quit <- struct{}{}发送一个空结构体就可以吗，请教下Free方式该如何写
第二个问题，demo1中好像也没看看到p.wg.Wait()

作者回复: 原文中有源码的链接，在最后。

源码在 https://github.com/bigwhite/publication/tree/master/column/timegeek/go-first-course/35

看了后，就可以回答你的问题了。

共 5 条评论 >

1

Darren

2022-01-17

老师以下几个问题哈：

1、第一种实现中，这块是不是有点问题：

```
go func() {
    defer func() {...
```

展开

作者回复: 看的真细致！

1. worker一旦创建后，除了panic和quit通知退出，worker是不会退出的，也就是没有所谓"正常退出"的情况。所以没在defer中调用<-p.active。
2. 的确是笔误,感谢指出。
3. 在文后有源码链接。这里的task仅是触发了worker创建，这里是调度循环，不处理task，所以要把task扔回tasks channel，等worker启动后再处理。

共 9 条评论 >

ㄟ骑着小车去兜风。

2022-01-17

看了这么长时间的专栏，感觉受益匪浅。之前虽然用过go但是一直没有系统的学习过，在开发过程中也遇到了许多坑。这段时间刷了两次函数方法和接口每次都能发掘到新的知识点，感觉对go有了全新的认识。对于今天这个小项目完全可以看的明白，但是如果说现在让我实现一个类似的协程池，就感觉自己没有这种思想头绪不知道从哪下手解决。我想问老师的是：怎么能够将自己所学的知识和实际的问题场景结合起来去解决实际问题，有...

展开

1

在下宝龙、

2022-01-17

最后一个例子，p := workerpool.New(5, workerpool.WithPreAllocWorkers(false),workerpool.WithBlock(false))
time.Sleep(time.Second * 2)
都给了两秒创建worker 都没创建好worker 吗？，按照代码的意思，即使不允许预先分配worker，作者run里面，继续分配worker 了啊，并且一开始没有task，直接就走default bre...

展开

作者回复: sleep的2s并没有创建worker啊。这里指定的WithPreAllocWorkers(false)，即不预创建。run阻塞在对tasks channel的读取上，直到后面main goroutine第一次调用Schedule写入一个task，这时run才会开始按需创建Worker。

共 3 条评论 >

1

无名无姓

2022-01-17

还是有些不理解

作者回复: 可以把问题提的具体一点😁。